VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Řízené nasazování JavaEE aplikací na aplikační server

Managed JavaEE Applications Deployment to Application Server

2015 Dominik Čech

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Dominik Čech**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Řízené nasazování JavaEE aplikací na aplikační server
Managed JavaEE Applications Deployment to Application Server

Zásady pro vypracování:

Cílem práce je vytvořit systém pro nasazování JavaEE aplikací na aplikační server JBoss (případně jiný), který umožní nasazování aplikací více uživatelům bez nutnosti přiřazovat jim administrátorská oprávnění. Zároveň bude systém umožňovat omezení velikosti nasazených aplikací pro konkrétního uživatele a definování prefixu cesty všech aplikací daného uživatele.

Systém bude umožňovat:
1. Autentizaci uživatelů pomocí LDAP serveru.
2. Nasazení EAR neb WAR aplikace pomocí webového rozhraní.
3. Sledování velikosti všech nasazených aplikací jednoho uživatele.
4. Omezení maximální velikosti nasazených aplikací uživatele.

Práce bude obsahovat:
1. Implementaci výše popsané JavaEE aplikace.
2. Programátorskou dokumentaci řešení s využitím diagramů jazyka UML.
3. Popis problematiky nasazování JavaEE aplikací.

Seznam doporučené odborné literatury:

[1] MARCHIONI, Francesco. JBoss AS 7 Configuration, Deployment and Administration. Birmingham: Packt Publishing, Limited. ISBN 18-495-1678-2.
[2] JBOSS. JBoss AS 7.0 Documentation [online]. [cit. 2012-10-05]. Dostupné z: https://docs.jboss.org/author/display/AS7/Documentation

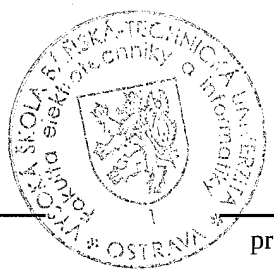
Dále podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2015

Tímto bych rád poděkoval všem, kteří mi s vypracováním práce pomohli a poskytli cenné podněty.

Abstrakt

Cílem této práce je vytvořit jednoduchý systém pro uživatelskou správu aplikací na JBoss aplikačním serveru, která je přístupná uživatelům z VŠB-TU Ostrava. Navržený systém je plně fukční a využívá JBoss Java API pro základní operace jak s aplikacemi tak s datovými zdroji. Systém je vhodný pro studijní účely nebo pro nasazení webových prezentací, napsané v Javě, na veřejný školní server.

Klíčová slova: JBoss 7.1, Java, Java EE 6, Application deploy, JBoss Java API

Abstract

Main goal of this work is to create simple system for maintaining user applications on JBoss application server, that is accesible for VŠB-TU Ostrava pupils. Designed system is fully functional and it uses JBoss Java API for basic application and datasource operations. System is suitable for educational purposes and for deploying web presentation, written with Java, to public school server.

Keywords: JBoss 7.1, Java, Java EE 6, Application deploy, JBoss Java API

Seznam použitých zkratek a symbolů

CLI	–	Command Line Interface
API	–	Application Programming Interface
HTTP	–	Hypertext Transfer Protocol
JNDI	–	Java Naming and Directory Interface
LDAP	–	Lightweight Directory Access Protocol
Ajax	–	Asynchronous JavaScript and XML
JTA	–	Java Transaction API
CCM	–	CORBA Component Model

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním cílem práce bylo vytvořit jednoduchý systém pro nasazování a správu aplikací na JBoss serveru a s tím související prozkoumaní možností JBoss serveru v otázce využití JBoss Java API pro komunikaci a správu jak aplikací tak datových zdrojů serveru.

V první části práce pojednává o JBoss serveru jako celku. Popisuje jakými způsoby lze daný server spravovat a jaké nám dává možnosti při nasazování aplikací. Poté se práce více zaměřuje na programovou stránku serveru, kde vysvětluje jakým způsobem lze komunikovat se serverem pomocí CLI operací a jejich intrepretace do Java kódu, také popisuje využití JBoss Java API pro práci s JBoss serverem. Práce obsahuje několik stručných příkladů, které ještě lépe vysvětlují danou tématiku. V poslední části popisuje samotnou aplikaci napsanou v Javě, která využívá výše zmíněné API pro komunikaci se serverem, dále také zmiňuje její funkce a celkový návrh aplikace.

2 Popis a správa JBoss serveru

JBoss server slouží jako aplikační server pro Java aplikace. Od verze 7.0 nabízí dva rozdílné módy. Domain mód, který umožňuje spustit a řídit více serverovou topologii z jednoho centralizovaného místa, a standalone mód, který vytvoří jednu nezávislou instaci serveru[2]. Je zcela na uživateli, který z módů si vybere.

Standalone mód se hodí pro většinu aplikací a i pomocí něho lze vytvořit více serverovou topologii, ale poté je na správci, aby koordinoval správu napříč všemi vytvořenými instancemi serveru. Na druhé straně domain mód je centralizovaný a správa všech instancí serveru lze provést z jednoho místa[2].

Spravovat nastavení a zdroje JBoss serveru lze třemi způsoby: použít webové rozhraní, příkazový řádek serveru nebo-li CLI nebo přímá editace konfiguračních xml souborů serveru. Ať už si uživatel zvolí jakýkoliv způsob nakonec se vše stejně synchronizuje napříč všemi pohledy a uloží do xml souborů[2].

2.1 Webové rozhraní

Webové rozhraní je jednoduchým a přehledným způsobem spravování serveru. Využívá HTTP management API pro správu domain nebo standalone serveru a je přístupné na obecné url adrese http://<host>:9990/console, výchozí nastavení využívá port 9990 a je přístupné na http://localhost:9990/console. JBoss AS ve verzi 7.1.x je distribuován s vychozím zabezpečením, proto je nutné vytvořit administrační účet a to pomocí skriptu add-user.bat (add-user.sh), který lze najít v bin složce JBoss serveru.

2.2 Editace xml souborů

Z mého pohledu se jedná o velice nepraktický a nepřehledný způsob správy. Uživatel by měl mít hluboké znalosti JBoss serveru a jeho konfiguračních souborů, aby byl schopen efektivně spravovat server tímto způsobem. V druhém případě by uživatel, který nemá takové znalosti mohl napáchat více škod než úžitku.

2.3 Command Line Interface

S trochou znalostí fungování a sestavování CLI příkazů, který popíšu v 2.3.1, se jedná asi o nejlepší a nejefektivnější způsob správy serveru. Využívá příkazové řádky serveru a řady možných příkazů a operací. Příkazovou řádku lze spustit pomocí bat nebo sh skriptu jboss-cli, který se nachází v bin složce JBoss serveru.

2.3.1 Struktůra CLI

V CLI můžeme použít buď to příkaz nebo operaci. Znalosti k této tématice čerpány z [3, 8].

2.3.1.1 Příkazy Většina příkazů se skládá z jedné nebo několika podoperací a slouží jako makra pro uživatele, mezi nejpoužívanější příkazy patří connect, deploy, undeploy a data-source. Jak už z názvu vyplívá connect se používá pro navázání spojení se serverem a jeho syntaxe je connect <host>:<port>, lze také napsat pouze connect což uživatele připojí na localhost:9999. Deploy a undeploy primárně slouží pro nasazování a odebírání aplikací ze serveru, ale také je možné tento příkaz použít např. pro nasazení ovladačů. Data-source je příkaz používaný pro vytvoření, smazání nebo modifikaci stávajícího datového zdroje. JBoss server samozřejmně nabízí více příkazů k použití, celý seznam lze vypsat pomocí příkazu help.

2.3.1.2 Operace Operace jsou nizkoúrovňovou interakcí se serverem, umožňují číst a modifikovat konfiguraci serveru, stejně jako byste modifikovali xml soubory přímo. Konfigurace je reprezentována stromem adresovatelných zdrojů serveru, kde každá větev stromu nebo-li zdroj nabízí určitou sadu operací, které lze na daném zdroji spustit. Operace se v podstatš skládá se tří částí: adresy, názvu operace a možnou sadou parametrů, které ne vždy jsou povinné.Jak už jsem dříve napsal konfigurace serveru je reprezentována stromem adresovatelných zdrojů, v tom případě kompletní adresa operace představuje cestu z kořene stromu nebo od stávající větve k cílovému zdroji a skládá se z páru klíč/hodnota, které jsou odděleny =. Jednotlivé větve jsou poté oddělovány /, například: /subsystem=datasources/data−source=.

Modifikace nebo čtení určitého zdroje provádí operace. Operace je textový řetězec začínající dvojtečkou. Každý zdroj má k dispozici operaci přidání s názvem „add“ a měl by mít i operaci s názvem „remove“. Operace „add“ slouží pro vytvoření daného zdroje a přijímá parametry, které jsou specifické pro daný zdroj. Na druhé straně operace „remove“ je bezparametrická a odstraňuje daný zdroj. Dále by každý zdroj měl mít k dispozici určitou sadu globálních operací. Seznam dostupných globalních operací najdete zde 2.3.1.3.

```
[/ node−type=node−name (/node−type=node−name)∗] : operation−name [( [parameter−name=
    parameter−value (,parameter−name=parameter−value)∗] )]
```

Výpis 1: Formální syntaxe operace

2.3.1.3 Výpis globálních operací

- read-resource
- read-attribute
- write-attribute
- read-resource-description
- read-operation-names
- read-operation-description

- read-children-types
- read-children-names
- read-children-resources

3 Nasazování aplikací na server

JBoss nabízí několik různých přístupů správy a nasazování aplikací na server. Jedná se o využití metod příme nazasaní aplikace, CLI, webové rozhraní serveru a programové nasazování aplikací.

3.1 Přímé nasazení aplikace

Jedná se o poměrně jednoduchý způsob nasazení aplikace, kde jedinou podmínkou je, aby uživatel měl přístup a oprávnění zapisovat do složky deployments v adresáři JBoss serveru. Pokud uživatel splňuje tuto podmínku poté stačí nahrát/nakopírovat daný soubor(jar, war, ear nebo sar), který chce nasadit, do této složky, soubor bude poté automaticky nasazen.

3.1.1 Deployment scanner

Je třeba ještě zmínit, že přímé automatické nasazování využívá tzv. deployment sacnner, který může fungovat ve dvou módech automatické nasazování nebo ruční nasazování[4].

3.1.2 Automatické nasazování

Automatické nasazování mód znamená, že deployment scanner přímo monitoruje nové soubory a automaticky nasazuje aplikace nebo provádí tzv. znovu nasazení když existující soubor nemá stejné časové známky[4]. Také vytváří tzv. značkovací soubory, které indikují v jakém stavu se soubor/aplikace nachází(nsazen, smazán, ...).

3.1.3 Ruční nasazování

Na druhé straně mód ruční nasazení čeká na uživatelův pokyn kdy by měl provést nějakou operaci. Pokyn k dané operaci uživatel dává vytvořením nebo smazáním určitých značkovacích souborů, které informují deployment scanner o dané akci.

3.1.4 Značkovací soubory

Soubory, které informují o stavu daného souboru/aplikace. Značkovací soubor má vždy stejné jméno jako soubor, který popisuje. Značkovací soubor souboru example.war, který je úspěšně nasazen by vypadal takto example.war.deployed. Kdyby uživatel chtěl například v ručním-módu nasadit aplikaci vytvořil by soubor example.war.dodeploy a deployment scanner by se postaral o zbytek.

3.2 Nasazení aplikace v CLI

CLI nabízí dva příkazy jeden pro nasazení a jeden pro odebrání aplikace ze serveru. Pro nasazení aplikace se používá příkaz deploy <cesta k souboru>. Jak už jsem zmínil dříve server tento příkaz rozdělí na dvě podoperace, jedna z nich je přidání souboru

do repozitáře serveru a druhá je zmíněné nasazení aplikace do runtime serveru. Příkaz deploy se může použít také pro nasazování ovladačů potřebné pro správné fungování datových zdrojů. Opakem příkazu deploy je příkaz undeploy, který určenou aplikaci odstraní z runtime.

3.3 Nasazení aplikace přes webové rozhraní

Poslední rychlou cestou nasazení aplikace je použití webového rozhraní. Jak se připojit do webového rozhraní jsem popsal v kapitole 2.1. Po připojení do webového rozhraní klikneme na Manage Deployments pod záložkou Deployments, zobrazí se nám tabulka všech nahraných aplikací, zde máme možnost nahrát novou aplikaci(Add content tlačítko), povolit či zakázat aplikaci nebo smazat. Při přidání nové aplikace nás dialogové okno vyzve k zadání aplikace, kterou chceme nahrát, vybereme a uložíme změny, poté je ještě třeba danou aplikaci povolit, aby proces nasazení byl kompletní. Správně nasazená aplikace by měla mít obrázek fajfky ve sloupci Enabled, pokud nemá je možné, že došlo k chybě. V tomto případě informace o chybě budou zobrazeny v pravém horním rohu pod odkazem Messages.

3.4 Využití JBoss Java API pro programové nasazování aplikací

JBoss Java API se využívá k programové správě JBoss serveru, abychom ho však mohli využít potřebujeme balíček s názvem jboss-as-controller-client pro naši verzi serveru. V mém případě se jedná o verzi 7.1.1 Final. Nejjednodušším způsobem zpřístupnění balíčku je využití správce návazností Maven, kde do našeho pom.xml souboru přidáme návaznost na daný balíček, Maven se poté postará o zbytek.

```
<dependency>
  <groupId>org.jboss.as</groupId>
  < artifactId >jboss-as-controller-client</artifactId>
  <version>7.1.1.Final</version>
</dependency>
```

Výpis 2: Přidaní závislosti do Maven pom xml

3.4.1 Připojení k serveru pomocí třídy ModelControllerClient

Jedná se o základní třídu, pomocí které se lze připojit a spravovat jboss server. K vytvoření klienta využijeme třídu ModelControllerClient.Factory. Příklad vytvoření klienta připojeného na localhost s portem 9999.

```
ModelControllerClient client = ModelControllerClient.Factory.create(InetAddress.getByName("
    localhost"), 9999);
```

Výpis 3: Vytvoření instance klienta

Tímto způsobem jsme vytvořili neověřeného klienta. V některých situacích chceme, aby klient byl ověřen podle jeho uživatelského jména a hesla, toho můžeme docílit předáním implementace rozhraní javax.security.auth.callback.CallbackHandler[1]. Implementace rozhraní a následné vytvoření ověřeného klienta by mohlo vypadat následovně. Úryvek kódu převzat z [1]

```
ModelControllerClient createClient(InetAddress host, int port, final String username, final String
      password) {

    CallbackHandler callbackHandler = new CallbackHandler() {
        @Override
        public void handle(Callback[] callbacks) throws IOException,
              UnsupportedCallbackException {
            for (Callback current : callbacks) {
                if (current instanceof NameCallback) {
                    NameCallback ncb = (NameCallback) current;
                    ncb.setName(username);
                } else if (current instanceof PasswordCallback) {
                    PasswordCallback pcb = (PasswordCallback) current;
                    pcb.setPassword(password.toCharArray());
                } else if (current instanceof RealmCallback) {
                    RealmCallback rcb = (RealmCallback) current;
                    rcb.setText(rcb.getDefaultText());
                } else {
                    throw new UnsupportedCallbackException(current);
                }
            }
        }
    };

    return ModelControllerClient.Factory.create(host, port, callbackHandler);
}
```

Výpis 4: Vytvoření ověřeného klienta

Je třeba ještě zmínit, že vytvořením ModelController klienta vám neřekne jestli je zapotřebí ověření nebo ne, až poté co spustíte operaci se dozvíte jestli je potřeba ověření[1]. Například v aplikaci kde chcete pracovat s ověřeným i neověřeným klientem, první vytvoříte neověřeného klienta , na kterém spustíte libovolnou operaci, pokud spuštění operace vyhodí vyjímku, zachytíte ji a požádate uživatele o zadání jména a hesla, poté můžete vytvořit ověřeného klienta. Implementace takového scénáře by mohla vypadat následovně.

```
ModelControllerClient client = null;

ModelControllerClient unauthenticatedClient = ModelControllerClient.Factory.create(InetAddress.
     getByName("localhost"), 9999);

try {
    ModelNode testConnection = new ModelNode();
    testConnection.get("operation").set("read-resource");

    unauthenticatedClient.execute(testConnection);
```

```
        client = unauthenticatedClient;
    } catch(Exception e) {
        client = createClient(InetAddress.getByName("localhost"), 9999, username, password);
    }
```

Výpis 5: Test jestli je zapotřebí ověřený klient

Poskládáním všech částí dostaneme mnou napsanou třídu ClientConnectionFactory, která se stará o vytváření klienta pro operace jako deploy, undeploy a další.

```
public enum ClientConnectionFactory {
    INSTANCE;

    private static final String HOST = "127.0.0.1";
    private static final int PORT = 9999;
    private static final String USERNAME = "administrator";
    private static final String PASSWORD = "admin";

    private InetAddress HOST_ADDR = null;

    private ClientConnectionFactory() {
        try {
            HOST_ADDR = InetAddress.getByName(HOST);
        } catch (UnknownHostException ex) {
            Logger.getLogger(ClientConnectionFactory.class.getName()).log(Level.SEVERE, null,
                ex);
        }
    }

    private ModelControllerClient createAuthenticatedClient(final String username, final String
        password) {
        CallbackHandler callbackHandler = new CallbackHandler() {
            @Override
            public void handle(Callback[] callbacks) throws IOException,
                UnsupportedCallbackException {
                for (Callback current : callbacks) {
                    if (current instanceof NameCallback) {
                        NameCallback ncb = (NameCallback) current;
                        ncb.setName(username);
                    } else if (current instanceof PasswordCallback) {
                        PasswordCallback pcb = (PasswordCallback) current;
                        pcb.setPassword(password.toCharArray());
                    } else if (current instanceof RealmCallback) {
                        RealmCallback rcb = (RealmCallback) current;
                        rcb.setText(rcb.getDefaultText());
                    } else {
                        throw new UnsupportedCallbackException(current);
                    }
                }
            }
        };

        return ModelControllerClient.Factory.create(HOST_ADDR, PORT, callbackHandler);
```

```
    }

    public ModelControllerClient createClient() {
        ModelControllerClient client = null;

        ModelControllerClient unauthenticatedClient = ModelControllerClient.Factory.create(
                HOST_ADDR, PORT);

        try {
            ModelNode testConnection = new ModelNode();
            testConnection.get("operation").set("read-resource");
            unauthenticatedClient.execute(testConnection);
            client = unauthenticatedClient;
        } catch(Exception e) {
            client = createAuthenticatedClient(USERNAME, PASSWORD);
        }

        return client;
    }
}
```

Výpis 6: Singleton ClientConnectionFactory

3.4.2 Vytváření CLI operací třídou ModelNode

Třída ModelNode je způsob jakým lze reprezentovat CLI operace posílané na server. Formát operace vyprodukovaný touto třídou se velice podobá JSON formátu a jeho struktůra se skláda ze tří částí. První z nich je operace, která určuje název operace, druhá část je adresa, která popisuje cestu k danému zdroji a poslední část jsou povinné nebo volitelné parametry operace.

```
{
    "operation" => "write-attribute",
    "address" => [
        ("subsystem" => "datasource"),
        ("data-source" => "name_of_datasource")
    ]
    "name" => "user-name",
    "value" => "admin"
}
```

Výpis 7: Struktůra ModelNode CLI operace

Tato ukázková operace nám říká, že chceme spustit operaci write-attribute na zdroji, který se nachází na adrese `/subsystem=datasource/data-source=name_of_datasource` kde přepíšeme atribut user-name na hodnotu admin. Vytvoření takové operace pomocí ModelNode by vypadalo následovně.

```
ModelNode request = new ModelNode();
request.get("operation").set("write-attribute");
ModelNode addr = request.get("address");
addr.add("subsystem", "datasource");
```

```
addr.add("data-source", "name_of_datasource");
request.get("name").set("user-name");
request.get("value").set("admin");
```

Výpis 8: Operace napsaná pomocí ModelNode třídy

3.4.2.1 Popis třídy ModelNode Každá instance třídy ModelNode je určitého typu tzv. ModelType. Při vytvoření nové instance, typ je nastaven na hodnotu ModelType.Undefined. Tento typ se dynamicky mění podle toho co do ModelNode vložíme např.

```
ModelNode node = new ModelNode();
node.getType() //returns ModelType.UNDEFINED
node.set(5) //ModelType = ModelType.INT
node.set(true) //ModelType = ModelType.BOOLEAN
node.set("ahoj") //ModelType = ModelType.STRING
```

Výpis 9: ModelNode typy

V příkladu jsou uvedeny jednoduché základní typy, které může ModelNode nabývat, existují ale také komplexnější typy jako LIST, PROPERTY a asi nejpoužívanější OBJECT.

ModelType.PROPERTY Typ PROPERTY je reprezentován třídou Property, která interně obsahuje pár String => ModelNode hodnotu.

```
Property property = new Property();
property.set("node", 5); //property name = node => value = ModelNode with type
    ModelType.INT with value 5
ModelNode pNode = new ModelNode();
pNode.set(property); //ModelType = ModelType.PROPERTY
```

Výpis 10: ModelNode property

ModelType.LIST LIST reprezentuje pole hodnot. Vytvoření ModelNode typu LIST lze pomocí metody add, která přidá hodnotu do pole.

ModelType.OBJECT Nejpoúživanější typ ModelNode. Interně reprezentovaný v podobě Map<String, ModelNode>. ModelNode se nastaví na typ OBJECT v okamžiku použití metody get(String), která vrátí instanci ModelNode pokud ji najde v příslušném kontejneru Map podle zadaného klíče, pokud ne přidá do kontejneru nový ModelNode typu UNDEFINED pod zadaným klíčem a tuto instanci i vrátí. Proto je třeba dát si pozor, protože metoda get nikdy nevrací null pokud neexistuje hodnota s daným klíčem.

```
ModelNode node = new ModelNode();
node.get(" first ");
node.toString(); //output => {" first " => undefined}
node.get(" first ").set(5);
node.toString(); //output => {" first " => 5}
```

Výpis 11: ModelNode objekt

Informace o třídě ModelNode a typech převzaty z [5].

3.4.3 Spuštění operace a vyhodnocení výsledku

Po úspěšném připojení popsané v kapitole 3.4.1 a vytvoření operace 3.4.2 je třeba operaci spustit a vyhodnotit výsledky. Operace se spustí pomocí metody client.execute(ModelNode op) a výsledek je vrácen v podobě ModelNode. Stejně jako popis operace musí dodržovat určitou podobu i výsledek operace má svou vlastní podobu. Každý výsledek operace obsahuje tzv. outcome, který informuje jestli se operace zdařila „success“, nezdařila „failed“ nebo byla zrušena „cancelled“. V případě úspěchu výsledek může, ale nemusí, obsahovat prvek s názvem „result“. V případě úspěšné operace read-resource by prvek „result“ obsahoval informace o daných zdrojích. V případě neúspěchu vysledek obvykle obsahuje prvek „failure-description“, který popisuje co se stalo špatně. V některých případech výsledek může obsahovat prvek „response-headers“, který nese dodatečné informace o operaci, většinou se jedná o hodnotu „operation-requires-reload“ což může znamenat, že operace potřebuje restartovat server, aby požadované změny mohly být zavedeny[1].

```
ModelControllerClient client;
//connection to server
...

ModelNode op = new ModelNode();
// setting some operation
...

ModelNode response = client.execute(op);

//handling response
ModelNode outcome = response.get("outcome");
if (outcome.asString().equals("failed")) {
    ModelNode failMsg = response.get("failure-description");
    System.out.println(failMsg.asString());
} else if (outcome.asString().equals("success")){
    //operation succeded
    //possible retrieve of result if any
}
```

Výpis 12: Příklad spuštění operace a zobrazení výsledku

3.4.4 Vytvoření akcí, plánu a následné spuštění

Jak lze vidět na výpisech 8 a 12, vytvoření a spuštění operace pomocí třídy ModelNode je přímočaré a relativně jednoduché, avšak v případě potřeby použít příkazy jako deploy, undeploy a další, které se skládají z několika operací JBoss nabízí sadu tříd, které celý proces značně zjednodušuje. Funkčnost následujících tříd zjištěna z [6].

DeploymentAction Jedná se o rozhraní, které definuje základní informace o akci, kterou chceme provést.

DeploymentPlan Rozhraní definující seznam DeploymentAction. Díky tomuto rozhraní můžeme definovat plán skládající se z několika akcí, místo ručního spouštění akcí za sebou.

DeploymentPlanBuilder Jak už z názvu vyplívá jedná se o rozhraní určené pro vytvoření DeploymentPlan. Obsahuje metody, které zjednodušují vytváření akcí a celého plánu. Mezi ně patří:

- **add** Vytvoří akci pro přidání aplikace do repozitáře serveru. Vrací objekt typu AddDeploymentPlanBuilder, který umožňuje pomocí metody andDeploy() vytvořit akci, která po úspěšném nahrání aplikace do repozitáře rovnou aplikaci nasadí do runtime.
- **deploy** Vytvoří akci pro nasazení aplikace do runtime. Vrací objekt typu DeploymentPlanBuilder.
- **undeploy** Vytvoří akci pro odebrání aplikace z runtime. Vrací objekt typu UndeployDeploymentPlanBuilder, který umožňuje pomocí metody andRemoveUndeployed() vytvořit akci, která odebere aplikaci z repozitáře serveru.
- **redeploy** Vytvoří akci, která danou aplikaci odebere z runtime a poté znovu nasadí do runtime.
- **replace** Vytvoří akci, která danou aplikaci nahradí a poté ji nasadí do runtime.
- **remove** Opak metody add, smaže danou aplikaci z repozitáře serveru.
- **build** Sestavý DeploymentPlan z daných akcí a vrátí jeho instanci.

ServerDeploymentManager Rozhraní definuje metody pro spuštění DeploymentPlan a pro vytvoření InitialDeploymentPlanBuilder, který slouží jako výchozí bod pro vytvoření nového plánu. K vytvoření instance typu ServerDeploymentManager musíme využít implementace tohoto rozhraní což je třída ModelControllerServerDeploymentManager.

Z popisů rozhraní se může zdát, že dané akce a jejich spouštění nemá nic společného s kapitolou 3.4.2 kde jsem popisoval práci s třídou ModelNode. Proto zde je urývek implementace metody převzaté z [6], která sestavuje operaci.

```
ModelNode op = new ModelNode();
op.get(OP).set(COMPOSITE);
op.get(OP_ADDR).setEmptyList();
ModelNode steps = op.get(STEPS);
steps.setEmptyList();
op.get(OPERATION_HEADERS, ROLLBACK_ON_RUNTIME_FAILURE).set(plan.isGlobalRollback
    ());
// FIXME deal with shutdown params

OperationBuilder builder = new OperationBuilder(op);

int stream = 0;
for (DeploymentActionImpl action : plan.getDeploymentActionImpls()) {
```

```
        ModelNode step = new ModelNode();
        String  uniqueName = action.getDeploymentUnitUniqueName();
        switch (action.getType())  {
        case ADD: {
            configureDeploymentOperation(step, ADD, uniqueName);
            step.get(RUNTIME_NAME).set(action.getNewContentFileName());
            builder.addInputStream(action.getContentStream());
            //step.get(INPUT_STREAM_INDEX).set(stream++);
            step.get(CONTENT).get(0).get(INPUT_STREAM_INDEX).set(stream++);
            break;
        }
        case DEPLOY: {
            configureDeploymentOperation(step, DEPLOYMENT_DEPLOY_OPERATION, uniqueName
                );
            break;
        }
        case FULL_REPLACE: {
            step.get(OP).set(DEPLOYMENT_FULL_REPLACE_OPERATION);
            step.get(OP_ADDR).setEmptyList();
            step.get(NAME).set(uniqueName);
            step.get(RUNTIME_NAME).set(action.getNewContentFileName());
            builder.addInputStream(action.getContentStream());
            step.get(CONTENT).get(0).get(INPUT_STREAM_INDEX).set(stream++);
            break;
        }
        case REDEPLOY: {
            configureDeploymentOperation(step, DEPLOYMENT_REDEPLOY_OPERATION,
                uniqueName);
            break;
        }
        case REMOVE: {
            configureDeploymentOperation(step, DEPLOYMENT_REMOVE_OPERATION, uniqueName
                );
            break;
        }
        case REPLACE: {
            step.get(OP).set(DEPLOYMENT_REPLACE_OPERATION);
            step.get(OP_ADDR).setEmptyList();
            step.get(NAME).set(uniqueName);
            step.get(TO_REPLACE).set(action.getReplacedDeploymentUnitUniqueName());
            break;
        }
        case UNDEPLOY: {
            configureDeploymentOperation(step, DEPLOYMENT_UNDEPLOY_OPERATION,
                uniqueName);
            break;
        }
        default:  {
            throw MESSAGES.unknownActionType(action.getType());
        }
        }
        steps.add(step);
    }
```

Výpis 13: Útržek kódu z implementace rozhraní ServerDeploymentManager

Z výpisu lze vidět, že se jedná o kompozitní operaci(operace složená z více podoperací), která je poté spuštěna proti serveru. Nakonec ještě krátka ukázka jak by vytvoření plánu a jeho spuštění mohlo vypadat.

```
//create connection to server
ModelControllerClient client = ModelControllerClient.Factory.create(host, port);
//create deployment manager
ServerDeploymentManager manager = new ModelControllerServerDeploymentManager(client);
//create initial deployment plan
DeploymentPlanBuilder planBuilder = manager.newDeploymentPlan();

//load your application
File application = new File("myapp.war");
//add to repository and deploy actions
DeploymentPlan plan = planBuilder.add(application).andDeploy().build();
//execute plan
Future<ServerDeploymentPlanResult> result = manager.execute(plan);
```

Výpis 14: Vytvoření plánu a jeho spuštění

3.4.5 Vyhodnocení výsledku plánu

Metoda execute rozhraní ServerDeploymentManager vrací výsledek v objektu typu Future<ServerDeploymentPlanResult>. Future je rozhraní, které reprezentuje výsledek asynchronních výpočtů. Výsledek lze získat pouze přes metodu get, která vrátí výsledek pouze tehdy pokud jsou výpočty dokončeny[9]. V našem případě Future obsahuje objekt ServerDeploymentPlanResult, výsledek asynchronně spuštěné operace a k jeho získání použijeme metodu get. K dispozici máme dvě metody get, jedna je bezparametrická a vrací výsledek po dokončení výpočtů a druha má dva parametry, limit jak dlouho se má čekat na výsledek a typ časové jednotky(sekundy, minuty, ...), pokud výsledek není vrácen do časového limitu, funkce vyhodí vyjímku TimeoutException.

ServerDeploymentPlanResult Jednoduché rozhraní, které zapouzdřuje výsledek spuštěného DeploymentPlan. Obsahuje pouze dvě metody.

getDeploymentPlanId Vrací id zapouzdřeného plánu.

getDeploymentActionResult Přijímá id provedené akce, která náleží danému plánu a vrací výsledek dané akce ve formě ServerDeploymentActionResult objektu.

ServerDeploymentActionResult Rozhraní zapouzřuje výsledek akce. Definuje výčtový typ Result, který obsahuje několik stavů provedené akce.

NOT_EXECUTED Akce nebyla spuštěna.

EXECUTED Akce byla úspěšně provedena.

CONFIGURATION_MODIFIED_REQUIRES_RESTART Akce byla úspěšně provedena, ale je zapotřebí restartovat server.

FAILED Akce se nezdařila.

ROLLED_BACK Akce byla úspěšně provedena, ale změny byly vráceny, kvůli chybě v některé z akcí v plánu.

Kromě výčtového typu rozhraní definuje několik metod.

getResult Vrátí výčtový typ Result, metoda nikdy nevrací null.

getDeploymentException Vrací objekt typu Throwable pokud během provádění akce nastala vyjímka, pokud ne vrací null.

getRollbackResult Pokud výsledek akce je Result.ROLLED_BACK nebo Result.FAILED a DeploymentPlan má povelen rollback, vrací výsledek rollback operace ve fromě objektu ServerDeploymentActionResult.

Pokud bychom navázali na výpis 14, kde jsme vytvořili a spustili plán, zpracování výsledku by mohlo vypadat následovně.

```
//variables plan and result from code snippet above
try {
    //wait max 1 minute for result, after that TimeoutException is raised
    ServerDeploymentPlanResult planResult = result.get(1, TimeUnit.MINUTES);

    //go through every action in plan and get action result
    for (DeploymentAction deploymentAction : plan.getDeploymentActions()) {
        ServerDeploymentActionResult actionResult = planResult.getDeploymentActionResult(
            deploymentAction.getId());

        //get result enum
        Result outcome = actionResult.getResult();

        //for later use if any exception occured
        Throwable exception = null;
        switch(outcome) {
            case EXECUTED:
                //do stuff
                break;
            case NOT_EXECUTED:
                //do stuff
                break;
            case ROLLED_BACK:
                //do stuff, possible to save exception if any
                if (actionResult.getRollbackResult() != null) {
                    exception = actionResult.getRollbackResult().getDeploymentException();
                }
                break;
            case FAILED:
                //do stuff, possible to save exception if any
                exception = actionResult.getDeploymentException();
                break;
```

```
                default:
                    //do stuff
            }

            if (exception != null) {
                Logger.getLogger(SomeClass.class.getName()).log(Level.SEVERE, null, exception);
            }
        }

    } catch (InterruptedException ex) {
        Logger.getLogger(SomeClass.class.getName()).log(Level.SEVERE, null, ex);
    } catch (ExecutionException ex) {
        Logger.getLogger(SomeClass.class.getName()).log(Level.SEVERE, null, ex);
    } catch (TimeoutException ex) {
        Logger.getLogger(SomeClass.class.getName()).log(Level.SEVERE, null, ex);
    }
```

Výpis 15: Zpracování výsledku plánu

4 JBoss datové zdroje

4.1 Správa

4.1.1 Webové rozhraní

Webové rozhraní nám ke správě datových zdrojů nabízí jednoduché statistiky již vytvořených datových zdrojů, které můžeme najít v záložce Runtime->Subsystem Metrics->Datasources, a také nám nabízí možnost vytvářet nové nebo editovat existující datové zdroje. Pro přídání nebo editaci datových zdrojů se musíme přepnout do záložky Profile(pravý horní roh), zde v záložce Connector klikneme na Datasources. V sekci Available Datasources můžeme vidět všechny vytvořené datové zdroje a jejich stav(povolen/zakázán). V sekci Selection jsou poté bližší informace o označeném datovém zdroji, zde je také možnost daný datový zdroj editovat. Při přidávání nového datového zdroje je vyžadováno tzv. JNDI-name, které musí být ve tvaru „java:jboss/datasources/název“, zbylé požadované položky jsou shodné jako v kapitole 5.3.3 kde popisuju přidávání datových zdrojů v mé aplikaci.

4.1.2 CLI operace

V kapitole 2.3 jsem popsal základní struktůru CLI operací a příkazů, čehož můžeme právě teď využít. Datové zdroje jsou prvky subsystému proto adresa pro tento zdroj vypadá následovně „/subsystem=datasources“. Na téhle adrese můžeme spustit několik operací, jednak globální operace(jejich výpis najdete v kapitole 2.3.1.3) a specifické operace pro tento zdroj jako get-installed-driver a installed-drivers-list. Pro představu operace pro výpis seznamu datových zdrojů by mohla vypadat takto.

```
// simple datasources list
/subsystem=datasources:read-children-names(child-type=data-source)

// datasources list with datasource configuration
/subsystem=datasources:read-children-resources(child-type=data-source)
```

Výpis 16: Příklad operací pro výpis seznamu datových zdrojů

Pokud bychom chtěli pracovat s jedním specifickým datovým zdrojem, tento přístup by nebyl moc vhodný, proto CLI API nabízí způsob jak adresovat jeden specifický zdroj, v našem případě datový zdroj. Adresa pro datový zdroj s jménem „DataTest“ by vypadala následovně „/subsystem=datasources/data-source=DataTest“. Stejně jako v minulém případě na této adrese můžeme provádět jak globální operace tak operace specifické pro tento zdroj. Nyní uvedu příklady globálních operací, které nám umožní číst a modifikovat datový zdroj.

```
// list of attributes and their values for datasource DataTest
/subsystem=datasources/data-source=DataTest:read-resource

// read the attribute "user-name" from datasource DataTest
/subsystem=datasources/data-source=DataTest:read-attribute(name=user-name)
```

```
//change attribute user-name
/subsystem=datasources/data-source=DataTest:write-attribute(name=user-name,value=test)
```

Výpis 17: Příklad globálních operací pro modifikaci datových zdrojů

Tímto jednoduchým způsobem můžeme měnit vlastnosti datových zdrojů. S datovými zdroji lze dělat více pomocí specifický operací mezi které patří:

enable/disable tyto operace povolují nabo zakazují daný datový zdroj

test-connection-in-pool operace umožňující otestovat spojení s databází

add operace pro přidání nového datového zdroje. Při vytváření nového datového zdroje do adresy zadáváme název vytvářeného zdroje. Dejme tomu, že chceme vytvořit datový zdroj s názvem „NewDatasrc“, adresa pro vytvoření takového zdroje by vypadala následovně „/subsystem=datasources/data-source=NewDataSrc:add(...)“. Operace add má požadované parametry mezi které patří:

connection-url url pro připojení ke zdroji dat

driver-name název ovladače, který definuje přístup k datům

jndi-name specifikuje JNDI název datového zdroje. Vždy by měl začínat java:jboss/datasources/

remove operace smaže datový zdroj na kterém je provedena

Informace z této kapitoly čerpány z [7].

4.1.3 Programově s využitím CLI operací

Programová správa datových zdrojů je v podstatě spojení již dvou popsaných kapitol, jedná se o předešlou kapitolu 4.1.2, kde popisuju základní práci s datovým zdrojem pomocí CLI operací, a kapitolu 3.4.2, která popisuje zbůsob převodu CLI operace do Java kódu pomocí třídy ModelNode. Spojením těchto dvou kapitol byste měli dostat ucelený obrázek o tom jak programově spravovat datové zdroje.

5 Aplikace

5.1 Obecný popis

Jedná se o aplikaci pro jednoduché nasazování java aplikací na aplikační server. Uživatel má k dispozici přehledné uživatelské rozhraní a do aplikace se přihlašuje pomocí svého školního uživatelského jména a hesla. Aplikace také nabízí jednoduchou správu uživatelů a jejich práv. Výčet všech dostupných funkcí najdete v kapitole 5.2 a popis jak je použít v kapitole 5.3.

5.2 Funkce

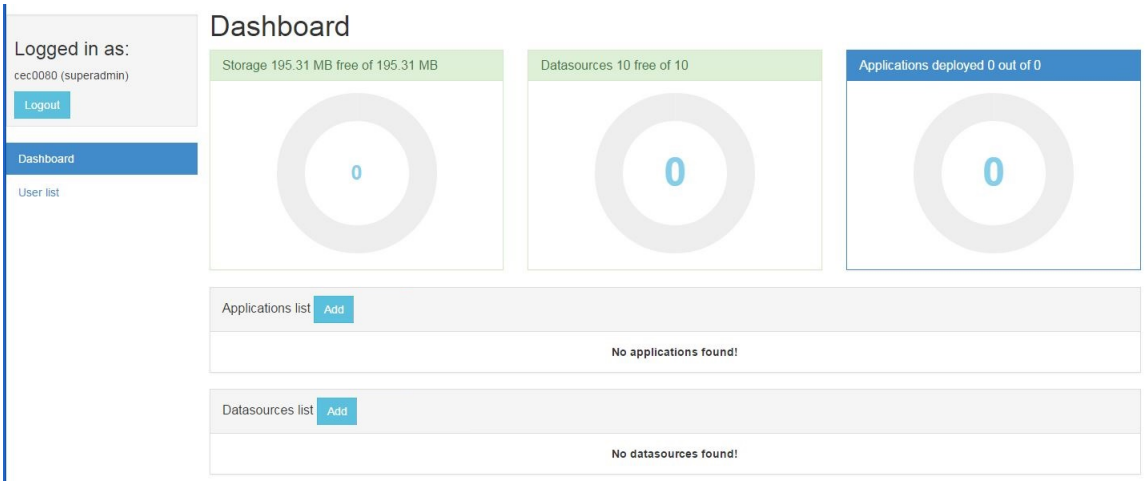
- příhlášení přes LDAP
- správa uživatelů(5.3.4)
  - rozdělení uživatelů do rolí(5.3.1)
  - omezení místa pro nahrávání java aplikací
  - omezení počtu vytvořených datových zdrojů
  - nastavování uživateslkých rolí
- správa aplikací(5.3.2)
  - podpora war, ear java aplikací
  - nasazení(deploy), znovu nasazení(redeploy), sesazení(undeploy), smazání(delete)
  - podrobný výpis chyby vzniklé při provádění operace nad aplikací
- správa datových zdrojů(5.3.3)
  - výtváření, testování spojení, editace, smazání
- základní nastavení přes konfigurační soubor(5.4)
- jednoduché uživateslké statistiky

5.3 Uživatelská příručka

Uživatelské rozhraní je navrženo tak, aby bylo co nejjednodušší a tím pádem rychlé pro práci. Jedná se o jednostránkový návrh kde uživatel může jednoduše kontrolovat své nahrané aplikace a vytvořené datové zdroje, také má k dispozici jednoduché statistiky, které zobrazují volný prostor pro aplikace, dostupný počet datových zdrojů k vytvoření a celkový počet nasazených aplikací.

5.3.1 Uživatelské role

Aplikace nabízí tři typy uživatelů superadmin, admin, uživatel.



Obrázek 1: Hlavní stránka aplikace

5.3.1.1 Uživatel Uživatelem se stává každý, při prvotním přihlášení do systému, pokud nemá nastavenou vyjímku v konfiguračním souboru, více v kapitole 5.4. Tento typ vidí pouze hlavní stránku a může provádět základní operace nad aplikacemi a datovými zdroji.

5.3.1.2 Admin Adminem se uživatel může stát dvěma způsoby, zvýšení pravomocí z uživatele na admina superadminem nebo adminem, nastavením přes konfigurační soubor. Stejně jako uživatel, admin má přístup k základním operacím, ale také i k seznamu všech uživatelů. V seznamu může vyhledat konkretní uživatele a upravit jim jejich základní nastavení účtu jako maximální prostor a počet datových zdrojů k vytvoření. Také může všem uživatelům nastavit jedním kliknutím defaultní hodnoty popsané v konfiguračním souboru nebo zvýšit pravomoce z uživatele na admina.

5.3.1.3 Superadmin Prvotní superadmin musí být nastaven v konfiguračním souboru před spuštěním aplikace nebo před prvotním přihlášením uživatele, který se má stát superadminem. Superadminem se uživatel může také stát navýšením pravomocí od jiného superadmina. Superadmin má absolutní pravomoce nad ostatními uživateli včetně adminů a superadminů, může odebírat a navyšovat práva, nastavovat omezení účtu, nastavovat defaultní konfigurační hodnoty a má možnost znovu nahrát konfigurační soubor.

5.3.2 Správa aplikací

Přidaní Přidat nebo-li nahrát novou aplikaci lze pomocí tlačítka Add na hlavní stránce. Nahrávaná aplikace musí splňovat několik podmínek, aby mohla být úspěšně nahrána.

1. Musí být v podporovaném formátu war nebo ear.
2. Pokud se jedná o soubor typu war, musí obsahovat soubor WEB-INF/jboss-web.xml s příslušným tágem context-root.
3. Pokud se jedná o soubor typu ear, musí obsahovat soubor META-INF/application.xml s příslušným tágem context-root.
4. Context-root aplikace musí být unikátní.
5. Název aplikace musí být unikátní.
6. Velikost aplikace se musí vejít do přiděleného místa uživatelského účtu.

Pokud aplikace splňuje tyto požadavky měla by být úspěšně nahrána. Systém automaticky mění název nahrávané aplikace na tvar login-název.typ.

Smazání Smazaní aplikace provedem kliknutím na tlačítko Delete v řádku aplikace, kterou chceme smazat. Pokud aplikace byla nasazena na serveru, automaticky se aplikace sesadí a smaže ze systému.

Nasazení Nasazení aplikace se provede kliknutím na tlačítko Deploy v řádku aplikace, kterou chceme nasadit na server. Nasazení aplikace může okamžik trvat. Po úspěšném nebo neúspěšném pokusu o nasazení se zobrazí informační zpráva o proběhnutí akce.

Znovu nasazení Znovu nasazení nebo-li Redeploy. Po kliknutí na tlačítko redeploy systém bude požadovat nahrát aplikaci. Nahrávaná aplikace musí mít stejný název a context-root jako aplikace, kterou se snažíme přehrát. Znovu nasazení neslouží pro přehrání kompletně jinou aplikací nýbrž stejnou aplikací se změnami od doby prvotního nahrání. Akce znovu nasazení v podstatě kombinuje několik akcí a to smazání staré verze aplikace, nahrání nové verze a nasazení nové verze. Jedná se o nejrychlejší způsob jak nasadit novou verzi aplikace.

Sesazení Sesazení aplikace nebo-li Undeploy slouží k sesazení aplikace z runtime.

Každá z těchto operací může skončit chybou, v tomto případě lze zobrazit celý výpis chyby kliknutím na odkaz v příslušném sloupci Deployment.

5.3.3 Správa datových zdrojů

Přidaní Stejně jako u aplikací, nový datový zdroj lze vytvořit kliknutím na tlačítko Add u tabulky Datasources. Následné okno bude požadovat několik informací o vytvářeném zdroji dat.

Name Jméno vytvořeného datového zdroje. Toto jméno musí být unikátní od všech vámi vytvořených datových zdrojů. Jméno se automaticky ukládá ve tvaru login-jméno.

Jndi-name Jméno pomocí, kterého vyhledáte daný datový zdroj ve vaší aplikaci. Také musí být unikátní a ukládá se ve tvaru java:jboss/datasources/login-jméno.

Driver Ovladač, který se bude používat ke komunikaci s vaší databází.

Connection url Url adresa vaší databáze.

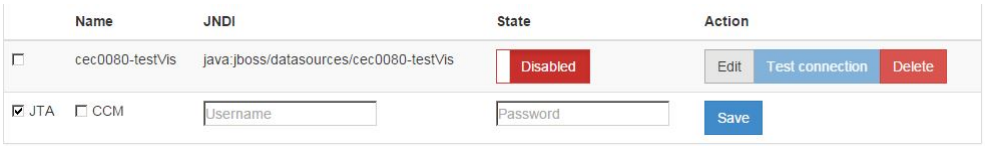
Username Uživatelské jméno k připojení do vaší databáze. Není povinné.

Password Heslo k připojení do vaší databáze. Není povinné.

Po úspěšném vytvoření se vám datový zdroj zobrazí v tabulce.

Smazání Smazaní datového zdroje provedeme kliknutím na tlačítko Delete. Pro správné smazání datového zdroje, zdroj nesmí používat žádná nasazená aplikace.

Editace Kliknutím na tlačítko Edit se zobrazí několik možností, které u datového zdroje můžeme upravit. Mezi ně patří použití JTA a CCM, úprava uživatelského jména a hesla k databázi. Změnu potvrdíme kliknutím na tlačítko Save. Editace datového zdroje na obrázku(2).



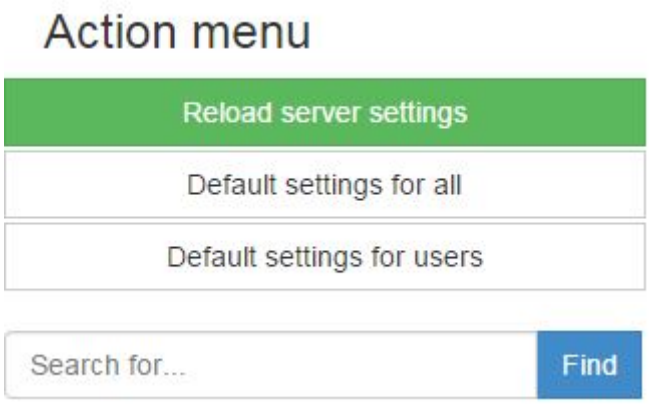
Obrázek 2: Editace datasourcu

Testování spojení Abyste mohli otestovat spojení, datový zdroj musí být ve stavu povolen(enabled).

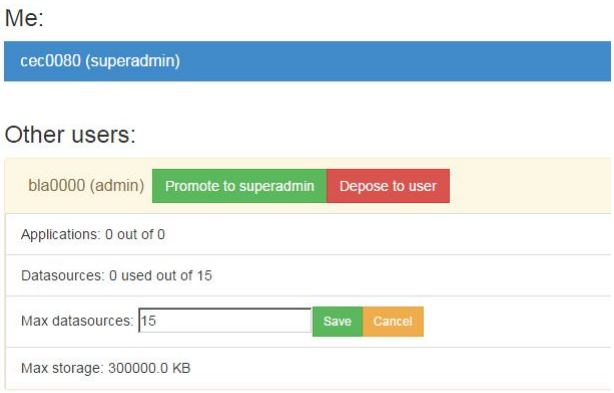
5.3.4 Správa uživatelů

Správu uživatelů mohou provádět pouze admini a superadmini. Superadmini narozdíl od adminů mnohou spravovat účty všech včetně jiných superadminů. Mezi pravomoce adminů patří editace maximálního přiděleného místa a maximální počet vytvořených datových zdrojů, také mají možnost jedním kliknutím změnit tyto hodnoty všem základním uživatelům na defaultní hodnoty uvedené v konfiguračním souboru. Superadmini mají navíc pravomoc znovu nahrát konfigurační soubor a jedním kliknutím hodnoty změnit na defaultní úplně všem.

Tato správa se nachází v uživatelském seznamu, po levé straně je Akční menu s danými možnostmi. Pokud chcete změnit hodnoty pouze jednomu uživateli, můžete využít vyhledávání podle loginu uživatele. Když uživatele najdete kliknete na jeho login a rozbalí se základní informace, poté stačí kliknout na hodnotu, kterou chcete změnit(pouze u maximálního místa a maximálního počtu datových zdrojů) a otevře se editační pole s tlačítky uložit a zavřít.



Obrázek 3: Superadmin menu akcí



Obrázek 4: Superadmin, seznam uživatelů s otevřenou editací

5.4 Xml konfigurační soubor

Konfigurační soubor bin/default_setting.xml slouží k základnímu nastavení systému. Definuje přístup k serveru, nastavení uživatelů a místo souborů kde se nachází context-root. Schéma souboru vypadá následovně.

```
<setting>
    <server>
        <host></host>
        <port></port>
        <login></login>
        <password></password>
        <applications-folder></applications-folder>
    </server>

    <default>
        <max-datasources></max-datasources>
        <max-storage></max-storage>
```

```
    </default>

    <exception id="" admin="">
        <max-datasources></max-datasources>
        <max-storage></max-storage>
    </exception>
</setting>
```

Výpis 18: Schéma konfiguračního souboru

host ip adresa serveru

port port na kterem se komunikuje se serverem, obvykle 9999

login login do administrace serveru

password heslo do administrace serveru

applications-folder složka kde se nahrávají aplikace, vždy by měla začínat .. pokud se složka nenachazi v bin adresáři serveru, složka by také měla obsahovat složku s názvem temp

default slouží pro nastavení defaultních hodnot všem uživatelům kromě těch, kteří mají nastavenou vyjímku

max-datasources maximální počet datových zdrojů k vytvoření

max-storage maximální prostor pro aplikace

exception definuje vyjímku pro určitého uživatele

exception parametr id login pro který se vyjímka vztahuje

exception parametr admin není povinný, slouží pro určení admina, 1 = superadmin, 2 = admin

Konfigurační soubor se nahrává při prvotním spuštění aplikace a poté ho může znovu nahrát pouze superadmin v administraci aplikace.

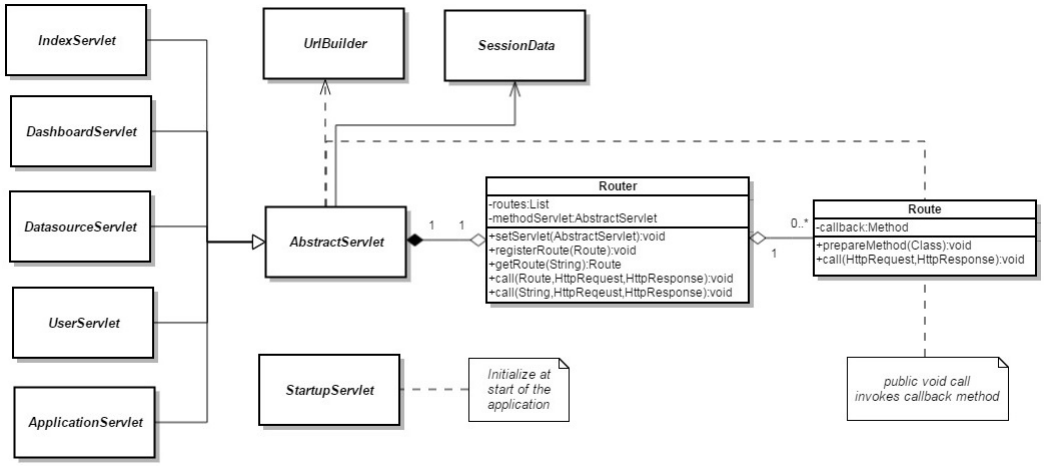
5.5 Popis návrhu a fungování

Systém byl navržena z ohledem na jednoduchost a snadnou rozšířitelnost. V následujících kapitolách se budu snažit popsat celou architekturu systému a jeho fungování.

5.5.1 Zpracování požadavku uživatele

O zachycení a další zpracování požadavku se starají tzv. Servlet třídy. Každá servlet třída má zaregistrované url adresy nebo url vzorce, podle kterých se pozná jaká třída má reagovat na jaký požadavek. Tyto url adresy a vzorce lze najít v konfiguračním souboru web.xml. Jeden z těchto vzorců je například /app/*, který má zaregistrovaná třída ApplicationServlet, což znamená, že když uživatel zašle například požadavek /app/deploy, zavolá se příslušná metoda třídy ApplicationServlet, která daný požadavek vyřídí. Samozřejmě uživatel může zadat různé požadavky jako /app/undeploy, /app/redeploy a další, v tomto případě je třeba rozlišit o jaký požadavek se přesně jedná a zavolat správnou metodu, tohle má na starosti směrovací systém, který se skládá ze dvou tříd Router a Route.

Třída Router je jednoduchou třídou pro zapouzdření seznamu instací třídy Route a nabízí rozhraní metod pro práci s tímto seznamem. Mezi nejdůležitější metody patří registerRoute, která přidává novou cestu do seznamu a poté metoda call, která volá přidělenou metodu k dané cestě. Třída Route popisuje metodu, která se má zavolat, když uživatel zašle požadavek. Ještě zde chybí poslední prvek k úspěšnému vyřízení požadavku, tento prvek je třída, která si uchovává instanci třídy Router a ve vhodnou chvíli volá příslušnou metodu pomocí třídy Route.



Obrázek 5: UML třídní diagram servlet části systému

Tato třída se jmenuje AbstractServlet a skoro všechny servlet třídy z ní dědí. AbstractServlet dědí z třídy HttpServlet a přepisuje metody doGet a doPost. Tyto dvě metody se starají o zachycení požadavku uživatele, poté co je požadavek uživatele zachycen metody zkonrolují jestli k danému požadavku existuje metoda a jestli je třeba zkontrolovat přihlášení uživatele. Jestli je vše v pořádku zavolá se příslušná metoda a požadavek je předán k vyřízení. Je třeba ještě zmínit, že každá servlet instance obsahuje svou vlastní instanci třídy Router a proto je třeba cesty registrovat, ve funkci init, pro každý servlet zvlášť. UML návrh servlet části lze vidět na obrázku(5).

Na obrázku(5) jsou znázorněny některé třídy, o kterých jsem se nezmiňoval. Třída UrlBuilder slouží k jednoduchému vytváření url odkazů a samotná třída AbstractServlet definuje metody composeLink a composeLinkAsString, které s touto třídou pracují. Také lze vidět třídu SessionData, která pouze zapouzdřuje data přihlášeného uživatele.

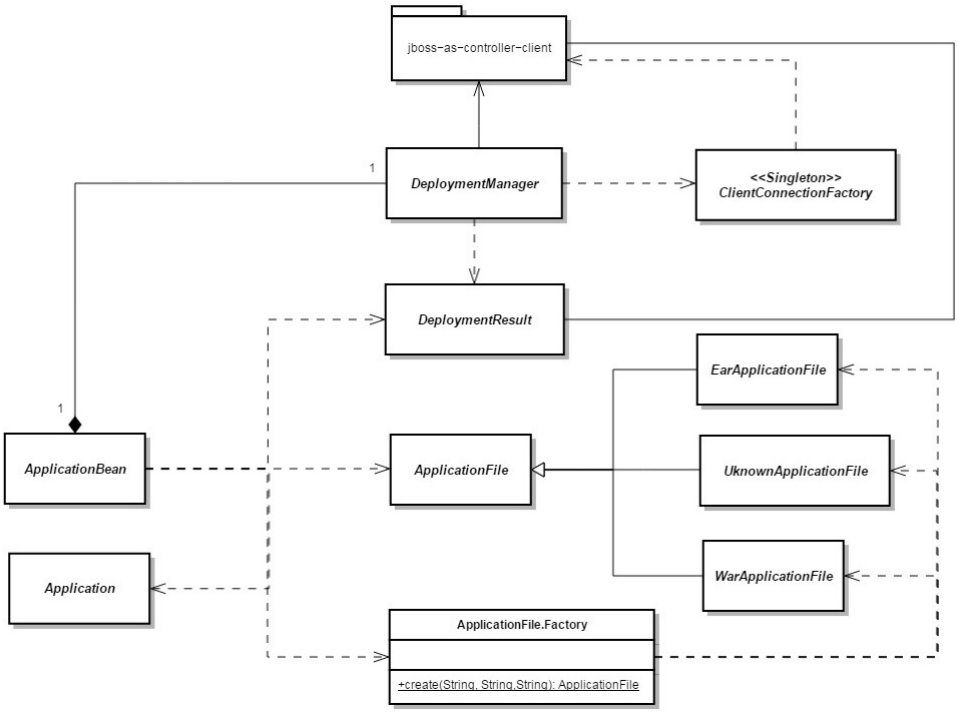
```
public class ApplicationServlet extends AbstractServlet {

    //some instance or class variables

    // init function is called only once, when the first request to this servlet arrives
    @Override
    public void init () {
        super.init () ; //important to initialize servlet config

        // tell router who is his owner
        router.setServlet(this) ;

        // register routes
        //you can use authorizedOnlyAccess or accessAll
        router.registerRoute(new Route("/app/upload", "uploadRequest").authorizedOnlyAccess());
        router.registerRoute(new Route("/app/deploy", "deployRequest").authorizedOnlyAccess());
        router.registerRoute(new Route("/app/redeploy", "redeployRequest").authorizedOnlyAccess
            ());
        router.registerRoute(new Route("/app/undeploy", "undeployRequest").
            authorizedOnlyAccess());
        router.registerRoute(new Route("/app/delete", "deleteRequest").authorizedOnlyAccess());
        router.registerRoute(new Route("/app/showFailMessage", "showFailMessageRequest").
            authorizedOnlyAccess());
    }

    public void uploadRequest(HttpRequest request, HttpResponse response){
      //some stuff
    }

    //other methods
}
```

Výpis 19: Příklad registrace cest a příslušných metod v třídě servlet

5.5.2 Práce s aplikacemi a návrh okolo třídy ApplicationBean

Třída ApplicationBean se stará o veškeré akce prováděné nad aplikacemi. Obsahuje metody pro nasazení, nahrání, smazání, sesazení a znovu nasazení aplikace na a ze serveru. Také se jedná o prostředníka mezi servletem a databází. Pro komunikaci s databází využívá EntityManager a doménový objekt Application, který obsahuje veškeré informace o jednom záznamu aplikace v databázi. Třída má mimo jiné také na starost kontrolu aplikace jestli splňuje všechny požadované kritéria. Kritéria, které aplikace musí splňovat můžete nají v kapitole 5.3.2. UML návrh systému kolem této třídy lze vidět na obrázku 6.

V návrhu lze vidět, že třída ApplicationBean používá pro svou práci také jiné pomocné třídy jako DeploymentManager, která hlavně slouží pro základní operace s apli-



Obrázek 6: UML třídní diagram návrhu práce s aplikacemi

kacemi a komunikaci se serverem, a poté ApplicationFile, která slouží pro nahrávání aplikace z lokálního disku na server a kontrolu validního kontextu aplikace.

ApplicationFile je abstraknı́ třída, která je momentálně děděná třemi dalšími třídami, WarApplicationFile, EarApplicationFile a UknownApplicationFile. Tyto třídy určují jaké typy systém dokáže zpracovat. Pro přidání nového typu se musí vytvořit nová třída, která bude dědit z ApplicationFile a bude implementovat dvě metody, isValid, která pouze vrací jestli se jedná o validní typ aplikace a getContextRootLocation, která vrací cestu k souboru kde se nachází kontext aplikace. Také je nutné v konstruktoru třídy zavolat init metody rodiče. Poté co je nová třída vytvořená a implementovaná je ještě zapotřebí přidat vytvoření instance do třídy ApplicationFile.Factory. Tato třída se stará o vytvoření správné instance podle typu souboru.

```
//create new class file for JarApplicationFile
public class JarApplicationFile extends ApplicationFile {
    public JarApplicationFile(String login, String filename) {
        super.init(login, filename); //call init function, for initializing variables
    }

    //override abstract methods from parent
    @Override
    public boolean isValid() {
        return true;
    }
```

```
    @Override
    public String getContextRootLocation() {
        return "path to xml where is context root";
    }

    //you can override hasValidContextRoot and getContextRoot, if application type doesnt need
        context root
}

//change Factory class in ApplicationFile
public static class Factory{
    public static ApplicationFile create(String login, String fileType, String filename) {
        switch(fileType.toLowerCase().trim()) {
            case "war": return new WarApplicationFile(login, filename);
            case "ear": return new EarApplicationFile(login, filename);

            //add case for your type
            case "jar": return new JarApplicationFile(login, filename);

            default: return new UknownApplicationFile();
        }
    }
}
```

Výpis 20: Příklad přidání nového podporovaného typu aplikace

Dále se posouváme k DeploymentManager třídě. Třida využívá k navázání spojení se serverem třídu ClientConnectionFactory, její implementaci můžete vidět ve výpisu 6. Také využívá třídy DeploymentResult, která slouží jako pomocná třída k zapouzdření výsledku operace. DeploymentManager využívá principů popsané v kapitole 3.4.4. Nasazení aplikace a uložení výsledku pomocí DeploymentManageru vypadá následovně.

```
public class DeploymentManager {
    //constants
    //how long we should wait for results
    private static final int WAIT_FOR_RESULTS = 1;

    //actions
    private static final String ADD_ACTION = "add";
    private static final String DEPLOY_ACTION = "deploy";
    private static final String UNDEPLOY_ACTION = "undeploy";
    private static final String REDEPLOY_ACTION = "full_replace";
    private static final String REMOVE_ACTION = "remove";

    private ModelControllerClient client = null;
    private ModelControllerClientServerDeploymentManager deploymentManager = null;

    private InitialDeploymentPlanBuilder deploymentPlanBuilder;

    public DeploymentResult deploy(String filename) throws IOException {
        client = ClientConnectionFactory.INSTANCE.createClient();
        deploymentManager = new ModelControllerClientServerDeploymentManager(client);
        deploymentPlanBuilder = deploymentManager.newDeploymentPlan();
```

```
        // this line is changing depending on action, deploy, redeploy, add, remove
        deploymentPlanBuilder = (InitialDeploymentPlanBuilder)deploymentPlanBuilder.deploy(
            filename);

        return executePlan(deploymentPlanBuilder.build(), DEPLOY_ACTION);
    }

    //execution of plan and gathering results in Map container
    private DeploymentResult executePlan(DeploymentPlan plan, String actionToReturn) throws
        IOException {
        try {
            Future<ServerDeploymentPlanResult> deploymentResult = deploymentManager.
                execute(plan);
            ServerDeploymentPlanResult result = deploymentResult.get(WAIT_FOR_RESULTS,
                TimeUnit.MINUTES);

            Map<String, DeploymentResult> deploymentResults = new HashMap<String,
                DeploymentResult>();
            for (DeploymentAction deploymentAction : plan.getDeploymentActions()) {
                String action = deploymentAction.getType().name().toLowerCase();
                deploymentResults.put(action, new DeploymentResult(action, result.
                    getDeploymentActionResult(deploymentAction.getId())));
            }

            return deploymentResults.get(actionToReturn);
        } catch (InterruptedException ex) {
            Logger.getLogger(DeploymentManager.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ExecutionException ex) {
            Logger.getLogger(DeploymentManager.class.getName()).log(Level.SEVERE, null, ex);
        } catch (TimeoutException ex) {
            Logger.getLogger(DeploymentManager.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            if (client != null) {
                client.close();
                client = null;
            }

            if (deploymentManager != null) {
                deploymentManager = null;
            }
        }

        return null;
    }

    //other methods for undeploy, add, redeploy and remove application, all uses the same
        principle as deploy method
}
```

Výpis 21: Příklad nasazení aplikace pomocí DeploymentManager třídy

Na výpisu si lze všimnout vytváření spojení pomocí ClientConnectionFactory a vytváření plánu, který se potom spouští v metodě executePlan. Nějakou dobu jsem si lámal

hlavu jakým způsobem ukládat výsledky z provedeného plánu, jelikož plán může obsahovat více jak jednu operaci, tak stejně bude obsahovat více jak jeden výsledek. Nakonec jsem se rozhodl výsledky zapouzdřit do třídy DeploymentResult a ukládat je do kontejneru typu Map, kde klíč je typ operace uložený v řetězci a k němu přísluší výsledek dané operace. Nakonec metoda executePlan vrací ten výsledek, který je specifikován parametrem actionToReturn. Po vrácení výsledku se režije předává zpátky ApplicationBean, který výsledek vyhodnotí a vrátí zprávu servletu jak operace dopadla.

5.5.3 Popis DatasourceBean

Třída DatasourceBean se stará jak už název napovídá o operace kolem datových zdrojů. Mezi ně patří vytváření, mazání, editace a testování spojení. Stejně jako DeploymentManager k připojení k serveru využívá ClientConnectionFactory. Pro vykonávání operací využívá principů popsaných v kapitole 4.1.2, v podstatě se jedná o vytvoření CLI operace, převedení této operace do ModelNode reprezentace, která se následně spustí proti serveru a vrátí výsledek. Výsledek operace se přeformuluje do objektu typu MessageResult, který je vrácen příslušnému servletu a ten poté podle typu požadavku(normální nebo Ajax) vypíše výsledek uživateli.

Nejzajmavější metoda třídy DatasourceBean je metoda pro editaci datového zdroje. Metoda je napsaná dynamickým stylem, což znamená že needituje natvrdo nadefinované položky ale přijímá kontejnér typu Map, kde klíč je název položky ve správné formě atributu datového zdroje a hodnota je hodnota daného atributu, která se nastaví. Stejným způsobem je řešen DatasourceServlet. Jediná úprava je nutna na straně zasílání požadavku, který se zasílá ajaxovým způsobem v souboru main.js. Je třeba zmínit, že názvy odeslaných polí v požadavku musí mít stejný název jako JBoss atributy datového zdroje, které se snažíte upravit, také posílaný požadavek by neměl obsahovat jiné data než id datového zdroje, uuid hash a editované položky, v případě že by bylo nutné, aby obsahoval i jiné data než ty, které se budou editovat, je zapotřebí upravit editRequest v třídě DatasourceServlet, aby tyto data přeskočil a nezasílal je do DatasourceBean.

5.5.4 UserBean a přihlašování uživatelů

Třída UserBean je poslední třídou, která pracuje s doménovým objektem a databází. Třída nabízí metody pro přihlášení uživatele do systému pomocí LDAP, stejně jako odhlášení. Také obsahuje metody pro kontrolu přihlášeného uživatele a metody pro akce popsané v kapitole 5.3.4, tyto metody pouze mění stav uživatele v databázi.

Jak už jsem napsal, přihlášení uživatele probíhá pomocí připojení na školní LDAP server, který ověří uživatelské jméno a heslo. Po úspěšném přihlášení se vytvoří nový uživatel v databázi pouze tehdy, pokud se uživatel do systému přihlásil poprvé a v databázi zatím není. Také se vygeneruje uuid hash klíč a uloží se do objektu SessionData, který se poté přenáší přes vygenerovaný session uživatele. Vygenerovaný hash klíč je nutné přenášet v každém požadavku jako uuid parametr. Tento parametr je poté při každém požadavku, který je zaslán na adresu s nastavenou autentizací, využit k porovnání vůči hash klíči uloženém v daném SessionData objektu. Proto je vhodné pro generování

url odkazů používat metody composeLink nebo composeLinkAsString, třídy AbstractServlet, které automaticky přidávají hash klíč do url adresy.

Doménový objek User, kromě dat o uživateli také obsahuje seznam všech aplikací a datových zdrojů, které uživatel vytvořil.

5.5.5 Třídy, o kterých jsem se nezmínil

V popisu architektůry mi zbýva popsat pár posledních tříd, o kterých jsem se zatím vůbec nezmínil. Mezi ně patří:

IndexServlet slouží jako výchozí bod pro celý systém. Stará se o zobrazení stránky pro přihlášení uživatele a některé servlety přesměrovávají uživatele na tuto stránku v případě, že není přihlášen a snaží se dostat někam kam by neměl.

DefaultSetting třída, která se stará o parsování konfiguračního xml souboru a ukládá si stav do proměnných. Parsování lze vyvolat metodou loadSetting. Jedná se o singleton a některé třídy využivají tuto třidu, aby se dostali k informacím o defaultních hodnotách uživatelů nebo systému.

StartupServlet volá se při prvním spuštění systému a je zodpovědná za prvotní nahrání konfigurace pomocí třídy DefaultSetting.

DahsboardServlet slouží k zobrazení výchozí administrace pro uživatele.

MessageResult zapozdřuje zprávy pro uživatele, které se zobrazují na stránkách.

6 Závěr

Závěrem bych chtěl říct, že navržený systém je fukční a využívá všech praktik popsané v této práci. Největším problémem bylo zjisti jak pracovat s JBoss serverem na programové úrovni, jelikož dobrých zdrojů informací o téhle tématice, které jsou na jednom místě, není mnoho.

Navržený systém je poměrně snadno rozšířitelný a je zde spoustu prostoru pro přidávání nových funkcí do systému, mezi které může například být komplexnější správa uživatelských rolí nebo větší možnosti v nastavení datových zdrojů.

7 Reference

[1] Brian Stansberry. The native management API – JBoss AS 7.1 – Documentation. Documentation – JBoss AS 7.1. [online]. 18.9.2012 [cit. 2015-04-27]. Dostupné z: *https://docs.jboss.org/author/display/AS71/The+native+management+API*

[2] Heiko Braun. Core management concepts – JBoss AS 7.1 – Documentation. Documentation – JBoss AS 7.1. [online]. 17.5.2013 [cit. 2015-04-27]. Dostupné z: *https://docs.jboss.org/author/display/AS71/Admin+Guide#AdminGuide-Coremanagementconcepts*

[3] Heiko Braun. Management resources – JBoss AS 7.1 – Documentation. Documentation – JBoss AS 7.1. [online]. 17.5.2013 [cit. 2015-04-27]. Dostupné z: *https://docs.jboss.org/author/display/AS71/Admin+Guide#AdminGuide-Managementresources*

[4] Heiko Braun. Application deployment – JBoss AS 7.1 – Documentation. Documentation – JBoss AS 7.1. [online]. 17.5.2013 [cit. 2015-04-27]. Dostupné z: *https://docs.jboss.org/author/display/AS71/Admin+Guide#AdminGuide-Applicationdeployment*

[5] Heiko Braun. Management API reference – JBoss AS 7.1 – Documentation. Documentation – JBoss AS 7.1. [online]. 17.5.2013 [cit. 2015-04-27]. Dostupné z: *https://docs.jboss.org/author/display/AS71/Admin+Guide#AdminGuide-ManagementAPIreference*

[6] GC: jboss-as-controller-7.1.1.Final.jar – GrepCode Java Project Source. GrepCode.com – Java Source Code Search 2.0. [online]. 9.3.2012 [cit. 2015-04-27]. Dostupné z: *http://grepcode.com/snapshot/repo1.maven.org/maven2/org.jboss.as/jboss-as-controller/7.1.1.Final*

[7] Stefano Maestri. How to create an manage datasources in AS7 | Planet JBoss Developer. JBoss Developer. [online]. 14.7.2011 [cit. 2015-04-27]. Dostupné z: *http://planet.jboss.org/post/how_to_create_an_manage_datasources_in_as7*

[8] Heiko Braun. Command Line Interface – JBoss AS 7.1 – Documentation. Documentation – JBoss AS 7.1. [online]. 17.5.2013 [cit. 2015-04-27]. Dostupné z: *https://docs.jboss.org/author/display/AS71/Admin+Guide/#AdminGuide-CommandLineInterface*

[9] Future (Java Platform SE 7). Java Platform SE 7 – Documentation. [online]. 26.9.2014 [cit. 2015-04-27]. Dostupné z: *http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html*

A Obsah CD

Na přiloženém CD jsou následující soubory:

- **/** - Text této bakalářské práce ve formátu PDF
- **/bin** - Zkompilovaná war aplikace s programy pro spuštění
- **/src** - Zdrojové kódy aplikace